An Algorithm for Optimal Back-Striding in Prolog

Vishv M Malhotra
School of Information Science and Technology
The Flinders University of South Australia
Bedford Park 5042 Australia.
vmm@cs.flinders.oz.au

Abstract

An intelligent backtracking algorithm to prune search space over which a Prolog interpreter searches a solution is given. The algorithm affects backtracking by determining, for each bound variable, a set of goals that can alter its binding. Under some real world constraints this backtracking is optimal. The algorithm keeps the overheads low by avoiding unnecessary enumeration of the sets.

1 Introduction

An execution of a Prolog program, under a conventional interpreter, is primarily controlled by its static text. Little use is made of the information that may be generated during the execution to prune search space. The onus of designing an efficient program is entirely on a programmer. The approach puts avoidable burden on the programmer and may adversely affect clarity of the program.

Several intelligent backtracking algorithms have been proposed [1, 4, 5, 6, 7] to improve efficiency of the Prolog interpreters. Typically, an algorithm prunes the search space of a program by selectively choosing the backtrack-points, the goals to which the interpreter backtracks. In this paper, a new algorithm to prune the search space is introduced. The main features of the algorithm are its selectivity, low overheads, and conceptual simplicity that extends naturally to programs with cuts if used within the framework of monotonic logic. The algorithm implements the deduction revision approach of Bruynooghe and Pereira [1] at a cost that is typical of the recently proposed low-overhead algorithms that sacrifice selectivity for implementation efficiency [4, 6]. These features make the algorithm a credible option for implementation on a standard interpreter.

In section 2, we briefly survey some of the other backtracking schemes and recapitulate the principles of intelligent backtracking. In section 3,

the basic idea underlying the proposed scheme is introduced. The basic algorithm is modified in Section 4 to reduce overheads. In section 5, we make some concluding remarks about the algorithm.

2 Principles of Intelligent Backtracking

A Prolog interpreter searches solutions for a program by executing goals in an orderly way. A goal is executed by replacing it with the body of a matching clause whose head can unify with the goal. We refer to this clause as the pairing clause of the goal; and, the goal as the parent of the goals in the body of the clause. A unifying substitution defines (generates) bindings for (some of) the variables. A search encounters a failure if the interpreter can not pair a goal with a matching clause. To continue a search after a failure the interpreter performs a backtrack by undoing some of the most recently executed goals. The (forward) search is resumed when a goal with an untried matching clause is reached. A typical interpreter makes no explicit effort to eliminate the reason for a failure. In rest of this paper, we shall use G to denote a goal and H to denote the head of a matching clause. H may or may not be unifiable with G.

It is well known [1, 3, 4, 5] that a non-unifiable subset of the executed goals plays a central role in avoiding backtrack-points that do not eliminate the reason for a failure. Suppose that during a search, G fails to unify with H. A set of goals in the search is non-unifiable if there is no successful search in which G pairs with H and the goals in the set have their pairing clauses as in the current search. We use S(G,H) to denote a non-unifiable set.

A search containing a non-unifiable set of goals can be terminated without losing a solution. An algorithm to systematically avoid these searches is given below. The algorithm determines for each goal a set of backtrack-points, B-list. When an interpreter fails to execute goal G using any of the matching clauses, the B-list of G is used to select the next backtrack-point. The algorithm, with minor variations, has been used in other intelligent backtracking schemes too [4, 6, 7].

A Backtrack Algorithm

For each goal G when it is created, along with its siblings to replace Parent(G), initialize B-list[G] := {Parent(G)}.

Whenever a goal, G, fails to unify with head H of a matching clause do the following:

1. *Compute B-list[G] := B-list[G] ∪ S(G,H). If G has no untried matching clause remaining then do the following (Steps 2 and 3):*

2. *Compute B-list[G] := B-list[G] - {G}.*

3. *Let M be the most recently executed goal in B-list[G]. Compute B-list[M] := B-list[M] ∪ B-list[G]. Finally, backtrack to M.*

Performance of the algorithm improves as the non-unifiable sets of the goals — S(G,H) — become smaller: backtracking to the destinations that do not eliminate the cause of the failure decreases as the sets become more selective. Many algorithms are known [1, 3, 4, 5, 6, 7] for determining non-unifiable sets of goals. Some algorithms require elaborate and complex analyses making them unsuited for an efficient implementation on a standard interpreter. Data dependency based schemes [2, 6, 7] overcome these difficulties by using simpler analyses. The B-list for a goal is determined by identifying the goals on which the relevant arguments (data) of the goal depend. The reported algorithms in this class, however, lack the selectivity of the other methods [1, 4].

Kumar and Lin [6] tag bound variables to remember goals generating the bindings. If a goal, G, fails, set S(G,H) is determined by identifying the goals that contribute, directly or transitively, a binding to the arguments of G. Set S(G,H) is determined solely by the arguments of G. As a result it is often large and lacks selectivity. An algorithm with a marginally better selectivity is given in [7]. A new algorithm with a substantially better selectivity is introduced in the next section. The algorithm improves the selectivity by identifying the data at the root of the failure.

3 The Proposed Scheme

The central idea in the algorithm is to determine, as an interpreter unifies two terms, a set of executed goals responsible for seeking their unification. A goal is responsible if the need to unify the terms would not have arisen if the bindings generated by the execution of the goal were not available. Should the unification fail, this set defines a non-unifiable set of goals. These goals if executed again, without changing any of the paired clauses, will necessarily repeat the failure. On the other hand, if the unification generates a binding for a variable, the set defines the goals inducing the binding. The variable would have remained unbound if the bindings generated by the goals in the set were not available. The set is called the determinant of the (bound) variable.

The recursive function Unify specifies the algorithm in a Pascal-like language using a call-by-value mechanism for passing parameters. The algorithm tags variables with their determinants as they are bound. The interpreter makes a top level call to the function to unify the corresponding

arguments of a calling goal and a matching head. These arguments constitute the first two parameters of the call. The third parameter specifies the set of goals responsible for seeking the unification of these parameters. Initially, the set contains the calling goal as the only member. During a unification whenever the algorithm traverses a bound variable, the set is augmented by including in it the determinant of the variable. These goals are indeed responsible for continuation of the process of unification over the term bound to the variable. The algorithm is similar to the method for obtaining inconsistent deduction trees [1].

```
function Unify (T1,T2: term; Responsibles: set_of_goals): set_of_goals;
                (* term(T) returns the term bound to variable T *)
  begin
     if T1 is an unbound variable
       then begin bind T2 to T1;          (* now, term(T1) = T2 *)
                  T1.Determinant := Responsibles;
                  return {}
            end ;
     if T2 is an unbound variable
       then return Unify (T2,T1,Responsibles);
     if T1 is a bound variable
       then return Unify (term(T1),T2,Responsibles ∪ T1.Determinant);
     if T2 is a bound variable
       then return Unify (T1,term(T2),Responsibles ∪ T2.Determinant);
     if T1.Functor ≠ T2.Functor          (* Not unifiable? *)
       then return Responsibles;          (* Return non-unifiable set.*)
       (* Else, unify the corresponding arguments of the functions *)
     for i := 1 to arity(T1) do
       begin R := i th argument of T1;
             S := i th argument of T2;
             if Unify (R,S,Responsibles) ≠ {}
                 then return Unify (R,S,Responsibles);
       end ;
     return {}          (*all arguments of the functions unify *)
  end;
```

An issue of importance that is not explicit in function Unify is the criterion for determining a variable to be unbound. Typically a unification of two (unbound) variables is handled by binding one of the variables to the other that remains free. We, however, view the variables as unbound

till one of them binds with a value. The motivation for departing from the convention becomes clear from the following example:

← f(R,S), g(S), h(S).
f(Z,Z).
g(100).
h(999).

Binding R, Z and S by letting Z and S point at R causes the unification of 100 and S, by goal g(S), to be expressed as binding between 100 and R. This obscures the fact that g(S) and h(S) constitute a non-unifiable set. The problem is overcome by directly binding the variables with their unifying terms (including other variables). Function IsFree implements the requisite definition of an unbound variable and provides a way for directly binding a variable with a unifying term. The function, when required, reverses (inverts) the direction of some of the bindings (pointers) among unbound variables to free the pointer associated with the unifying variable. A unification between a variable in a goal and a variable in the head of its pairing clause makes the variables equivalent. This equivalence is often implemented by binding a variable to the other. An equivalence binding does not need to be reversed. Likewise, a sequence of equivalence bindings may be replaced by a single binding to improve efficiency. These improvements are not incorporated in function IsFree to keep its description simple. Function Unify may use call IsFree(X,term(X)) to determine if X is an unbound variable.

```
function IsFree (X,Y:term): boolean;
begin
        if Y is nil
           then return true ;                (* X is unbound. *)
         (* Advice on restoring state of the bindings during backtrack-
           ing: Save address of X in trail stack with a special flag.
           Use call IsFree(X,term(X)) to undo changes during untrail*)
       if Y is not a variable
         then return false;          (* X is bound *)
       if IsFree (Y,term(Y))
         then begin (* Reverse binding. Need not trail the change *)
                    bind X to Y;              (* set term(Y) = X *)
                    Y.Determinant := X.Determinant;
                    return true
              end
          else return false
end;
```

Performance: Selectivity

To test selectivity of the algorithm in choosing backtrack-points, we used a simple procedure for placing queens on a 6×6 chess board similar to the one given in Bruynooghe and Pereira [1]: systematically generate configurations of the queens till an acceptable configuration is found. The number of configurations generated before the first solution is used as a measure of the search space size. We obtained about 75% (47 configurations instead of 187) reduction in the search space using the algorithm. The corresponding gains using either the algorithm due to Kumar and Lin [6] or Malhotra, To and Kanchanasut [7] is only about 1%.

Indeed the non-unifiable sets of goals, determined by the algorithm, are minimal under the following conditions:

(a) the process of unification terminates as soon as the terms are determined to be non-unifiable (i.e., only one reason for non-unification of G and H is used to determine S(G,H).), and,
(b) the goals are completely ignorant about their untried matching clauses (i.e., the future behavior of the program can not be predicted.).

Under these conditions one may conceive, for any goal in a S(G,H), a matching clause that does not bind any variable. The clause eliminates the (detected) reason for non-unification of G and H. (Bindings generated by different goals are distinct even if they bind the same term to a variable.) As a consequence no goal in S(G,H) may be omitted while backtracking to eliminate the reason for non-unification of G and H. This observation implies that the algorithm is optimal in the class of practical intelligent backtracking algorithms including DIB [4], sDIB [4], and data-dependency-based algorithms [6, 7].

Cuts and Multiple Solutions

Cuts and searches for multiple solutions of a program cause backtrackings for reasons other than unification failures. A cut may induce a backtrack from its parent goal. Search for the next solution of a program begins with a backtrack from a fictitious goal — a goal containing all answer variables of the user query and executed after the last goal of the successful search.

The purpose of these backtracks is to alter some (any) binding in the arguments of the goal. The union of the determinants of the (bound) variables in the arguments of a goal is precisely the set of goals capable of altering the bindings of the arguments. Thus, a backtrack due to these reasons is handled, in the proposed scheme, by including the determinants of the bound variables in the arguments of the goal into its B-list.

The method, however, is not usable with a non-monotonic construct —

that is, a negative goal or the parent goal of a cut used for non-monotonic reasoning — especially if a path exists in the program on which the goal may be executed with non ground arguments. To handle these constructs the algorithm needs to determine, for each unbound variable, an auxiliary determinant: the goals that receive a term containing the variable as an argument and therefore can bind the variable. Admittedly, the problem is far from solved and is not pursued further in this paper. An example of cut used for non-monotonic reasoning is given below. Notice that fail_if_not_generated(X) fails, but fail_if_not_generated(2) succeeds.

```
← genenerate(X), fail_if_not_generated(X).
generate(X).    /* This clause does not generate a value for X
generate(2).    /* This one does generate a value for X
test(9,no).
test(2,yes).
fail_if_not_generated(X) ← test(X,Y), !, set(Y).
set(yes).
```

4 A Practical Algorithm

The algorithm substantially reduces the search space but has considerable overheads. Each bound variable is tagged with its determinant. Much of this information remains unused — in any program, only some of the bindings cause failures. This penalty is avoided by keeping the determinants implicit during the forward execution of a program. In fact, overheads are minimized by identifying only one element of a set at a time.

Explicit enumeration of a determinant is avoided by assigning three tags, providing necessary information, with each bound variable: Caller, Var_Term and Val_Term. Caller identifies the calling goal that generated the binding. The other two tags identify the terms whose unification generated the binding. Var_Term is a pointer to the term containing the variable. Val_Term points to the term containing the bounded term. (A reader familiar with a WAM [8] would realize later that it is not essential to preserve two pointers to the terms. It suffices to remember the position of an argument in the head literal.)

The other change that we make in the algorithm is the way we define and handle a B-list. Now, a B-list is a set of bound variables. The determinants of these variables define the set of backtrack-points — a B-list as used in the previous sections. For a call, let terms T1 and T2 fail to unify because their (respective) subterms S1 and S2 are different. We include in the B-list of the calling goal all those bound variables that occur on the paths from

the root of T1 to S1 and from the root of T2 to S2. The modified function Unify incorporating these changes is given below:

```
function Unify(T1,T2:term; P1,P2:term): boolean;
        (* For the top level call T1 (P1 is a pointer to T1) and T2 (P2 is
          a pointer to T2) are the corresponding argument of the
          calling goal, G, and a matching head. G is a global parameter.*)
begin if IsFree(T1,term(T1))
            then begin bind T2 to T1;              (* term(T1) = T2 *)
                       T1.Caller := G; T1.Var_Term := P1;
                       T1.Val_Term := P2; return true
                 end ;
       if IsFree(T2,term(T2))
            then return Unify(T2,T1,P2,P1);
       if T1 is a bound variable
            then begin if Unify(term(T1),T2,P1,P2)
                              then return true
                         else begin   (* Subterms were not unifiable *)
                                    insert T1 in B-list[G];
                                    return false
                                  end
                 end ;
       if T2 is a bound variable
            then return Unify(T2,T1,P2,P1);
       if T1.Functor ≠ T2.Functor          (* unification fails *)
            then return false ;
       (* Try to unify the corresponding arguments of the functions *)
       for i := 1 to arity(T1) do
            begin R := i th argument of T1;
                  S := i th argument of T2;
                  if not Unify(R,S,P1,P2)
                       then return false
             end ;
       return true
end .
```

A backtrack-point is determined by examining Caller tags of the variables in an appropriate B-list. For any bound variable, the tag (Caller) specifies the most recently executed goal in the determinant of the variable. Thus, the most recently executed goal among these tags determines the backtrack-point.

Once the backtrack-point has been determined, the determinants of the bound variables whose Caller tags match the chosen backtrack-point are enumerated to expose other members of the sets. This is done by replacing each of these variables by a set of bound variables. The replacing variables are selected by using the tags Var_Term and Val_Term of the variables being replaced. The bound variables on the corresponding paths from the root of the term pointed to by Var_Term to the variable and the root of the term pointed to by Val_Term to the bound term constitute the replacing variables. After replacement of the variables, the B-list is merged with the B-list of the backtrack-point. If no backtrack is executed from the destination of the current backtrack, much of the effort will remain unused. It is therefore prudent to defer the replacement of the variables by preserving the necessary references. The B-list for a goal need be determined only when a backtrack from the goal is to be executed. Some paths are affected as a goal changes its pairing clause. In particular, a goal that has paired with a clause with multiple occurrences of a variable in its head may fragment a path as its pairing is undone during backtracking. In this case the replacement of the variables is best done before backtracking to the goal. A functional description of the backtrack algorithm incorporating these changes is given below:

A Modified Backtrack Algorithm

For each goal G, when it is created, initialize B-list[G] = {NEW_VAR}, where NEW_VAR is a new variable and NEW_VAR.Caller = Parent(G).

Whenever a goal, G, fails to unify with head H of a matching clause do the following:

1. *(Function Unify includes the appropriate bound variables in the B-list.) If G has no untried clause remaining then do the following (Steps 2 thru' 5):*

    2. *For each variable, V, in B-list[G] such that V.Caller = G do*

        a) remove V from B-list[G];

        b) include in B-list[G] all bound variable on the path from V.Var_Term to V in term V.Var_Term;

        c) include in B-list[G] all bound variable on the path from V.Val_Term to term(V) in term V.Val_Term;

    3. *Let M be the most recently executed goal among V.Caller, where V is a variable in B-list[G]. If the clause pairing M does not have multiple occurrences of any variable in its head then goto step 5.*

    4. *For each variable, V, in B-list[G] such that V.Caller = M do*

a) remove V from B-list[G];

b) include in B-list[G] all bound variable on the path from V.Var_Term to V in term V.Var_Term;

c) include in B-list[G] all bound variable on the path from V.Val_Term to term(V) in term V.Val_Term;

5. Merge B-list[G] into B-list[M] and backtrack to M.

Performance: Overheads

Overheads introduced by the algorithm are estimated using the study due to by Kumar and Lin [6]. They identified, for an implementation of their algorithm on a WAM [8], five components of the overheads: tagging, creating special choice-point, manipulating B-lists, traversing arguments, and, miscellaneous costs. The first two components specify the overheads in the forward execution of a program. The next two specify the overheads during backtracking. A new component of the overheads, introduced by the proposed algorithm, stems from the rearrangement of the bindings by function IsFree.

Shorter B-lists and limited traversals of the arguments of the goals would reduce the overheads in backtracking in the proposed algorithm over those reported in [6]. A naive algorithm may traverse arguments (in step 2 or 4 of the modified backtrack algorithm) by simply repeating the unification (actually, a simplified form) of the terms pointed to by Var_Term and Val_Term of a variable being replaced. Other components of the overheads identified in [6] are expected to remain unaltered in the proposed algorithm. The overhead due to function IsFree will be small. The function can only affect a binding that is generated by a variable with multiple occurrences in a clause head. Typically, only a few clauses have such variables. Also, a sequence of these bindings does not usually form a long chain. Indeed it is possible to avoid the use of function IsFree by foregoing the optimality of the algorithm. Only a few programs will be affected by this change.

Notwithstanding low overheads, the algorithm due to Kumar and Lin slows the executions of certain programs. This is attributed to the poor selectivity of the algorithm in choosing the backtrack points. The proposed algorithm promises better speed-up of an interpreter than the other schemes for the following reasons:

(a) The overheads in the forward execution of an interpreter are low. And,

(b) the algorithm is very selective in choosing the backtrack-points.

The former observation ensures that a program that does not involve extensive backtracking is not penalized. The latter observation benefits a

program with backtracking in two ways. Firstly, the interpreter needs to execute a smaller number of goals. Secondly, it reduces the fraction of the cost incurred on overheads. In any non-trivial intelligent backtracking scheme the backtrack steps are the dominant component of the overheads. A selective scheme benefits because it decreases the fraction of calls that shall require backtracking by eliminating many of these calls.

5 Concluding Remarks

The algorithm makes some interesting comparisons with other data dependency based backtracking algorithms. Chang and Despain [2] construct a dependency graph at compile time. Should a search fail, this dependency information is used to affect backtracking. The backtrack-points are selected based on their potential to contribute data to the failed goal. Kumar and Lin [6] select backtrack-points by collecting the dependency information during execution. A backtrack-point is selected from the goals actually contributing some data to the failed goal. The algorithm strives to keep the overheads low and performs no analysis to identify the culprit data. The algorithm may backtrack to a goal that has not generated data causing the failure. The present algorithm fills this lacuna. It performs an analysis to identify data at the heart of a failure. The dependency information is used to determine the goals generating the data. It is also interesting to note that DIB [4] too maintains detailed data about the various dependencies. The practical version of the algorithm sDIB [4], however, is not optimal. The algorithm reported in this paper is optimal yet has overheads similar to those of sDIB.

During our experiments using the queens problem we noticed certain resemblances between the patterns of queen placements under the proposed backtracking scheme and the clever solution given by Bruynooghe and Pereira [1]. It soon became clear that the resemblance is not coincidental! The cleverness incorporated in the program improves performance by avoiding superfluous searches. Certainly, it is convenient to let the interpreter prune the search space rather than to require a programmer to modify it.

References

1. *M. Bruynooghe and L.M. Pereira, Deduction Revision by Intelligent Backtracking, In: J.A. Campbell (ed.), Implementation of Prolog, Ellis Horwood, 1984, pp. 194-215.*

2. *J.-H. Chang and A.M. Despain, Semi-Intelligent Backtracking of Prolog Based on a Static Data Dependency Analysis, Proc. of IEEE*

Symp. Logic Programming, Aug., 1985, pp. 10-21.

3. *T.Y. Chen, J.L. Lassez and G. Port, Maximal Unifiable Subsets and Minimal Non-unifiable Subsets, New Generation Computing, Vol. 4, 1986, pp. 133-152.*

4. *C. Codognet, P. Codognet and G. File, Yet Another Intelligent Backtracking Method, In: R.A. Kowalski and K.A. Bowen (eds.), Logic Programming: Proc. 5th Intl. Conf. and Symp., MIT Press, Cambridge, Ma, 1988, pp. 447-465.*

5. *P.T. Cox, Finding Backtracking Points for Intelligent Backtracking, In: J.A. Campbell (ed.) , Implementation of Prolog, Ellis Horwood, 1984, pp. 216-233.*

6. *V. Kumar and Y-J. Lin, A Data-Dependency-Based Intelligent Backtracking Scheme for Prolog, J. Of Logic Programming, Vol. 5, Nr. 2,June 1988, pp.165-181.*

7. *V.M. Malhotra, T.V. To and K. Kanchanasut, An Improved Data-Dependency-Based Backtracking Scheme for Prolog, Information Processing Letters, Vol. 31, No. 4, May 22, 1989, pp 185-189.*

8. *D.H.D. Warren, An Abstract Prolog Instruction Set, Tech. Note 309, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025 (Oct., 1983).*

Proceedings of the 7th International Conference on Logic Programming (ICLP)
MIT Press. ISBN: 0-262-73090-1
Editors: DHD Warren & P Szeredi